GCC instruction scheduler and software pipelining on the Itanium platform

Arutyun Avetisyan, Andrey Belevantsev, and Dmitry Melnik *

Abstract

This paper describes the recent progress in implementing the global instruction scheduler and software pipeliner targeted at VLIW platforms for the GCC compiler. The scheduler is based on a selective scheduling approach, and a number of changes and improvements were made to make it feasible for a production environment and to improve code generation: speculation support, better instruction priorities, and heuristics to limit register renaming and pipelining code motion. Besides this, the paper outlines the improvements made to the Itanium backend and to the modulo scheduling implementation of GCC. Experimental results on SPEC FP 2000 benchmarks for the implemented scheduler and modified modulo scheduling are also presented.

1 Introduction

The Itanium platform is very demanding on the compiler quality. Its specific features, such as speculation, predication, and rotating registers require the full support from the compiler to be utilized. The GNU Compiler Collection (GCC) is the most popular open source compiler for Itanium, and its effectiveness directly influences Itanium's adaptation on GNU/Linux systems. Unfortunately, there is not much work going on to improve GCC's performance on Itanium.

During the last years, we have worked on several projects on adding to GCC the features that would be useful on Itanium. The latest project is implementing an aggressive interblock instruction scheduler for GCC, using the selective scheduling approach as a basis. The scheduler supports a number of instruction transformations (such as register renaming, forward substitution, bookkeeping code creation, and instruction unification), software pipelining of innermost loops, and IA-64 speculation capabilities. The implementation is available on the sel-sched branch in the GCC repository.

To make the implementation feasible for a production compiler, we needed to make a number of changes to the original approach, including handling instructions such as calls / inline assembly, support for speculation, enhancing choosing heuristics, and improving compile-time. We have also made a number of improvements to the Itanium backend of GCC. We discuss the changes we made and experimental results. Also, we outline the changes recently made to the modulo scheduling implementation of GCC to make it work on Itanium.

The rest of the paper is organized as follows. Section 2 provides an overview of the selective scheduling approach, which is needed for the following discussion. Section 3 discusses the GCC implementation and its improvements made by us, including managing of additional transformations. Section 4 discusses the efforts needed for improving compile-time performance of the scheduler. Section 5 outlines other improvements

*Institute for System Programming of RAS. {arut, abel, dm}@ispras.ru

made to the Itanium backend and presents experimental results for the selective scheduler in GCC. Section 6 outlines the changes made to the modulo scheduling implementation of GCC and presents performance data, while section 7 concludes.

2 The Selective Scheduling Approach

Selective scheduling is a global top-down scheduler approach operating on arbitrary acyclic regions with support for code motion with bookkeeping code creation. Each iteration of the scheduling loop gathers the group of instructions that may be executed in parallel at each *fence*, which represents the current scheduling point. The group represents a single VLIW instruction. There could be several active fences at once.

Scheduling an instruction into the parallel group consists of three steps. First, the set of available expressions is computed at the boundaries of the group by traversing the the DAG starting from the boundaries in reverse topological order. When visiting an instruction, the set of expressions available below it is propagated through this instruction, filtering out dependent expression, and then the instruction is added to the resulting set (like below). Intermediate availability sets are saved at the beginning of each basic block. Expression unification happens when availability sets of a branch point's successors are joined into one before propagating. When an expression is not available on some of the successors, its *spec* attribute is increased by one.

$$avset(n) = moveup\_set(\bigcup_{x \in Succ(n)} avset(x)) \bigcup av\_op(n)$$

Second, the best available expression is chosen for scheduling. The original approach suggests using the degree of speculativeness of each expression (calculated as the spec attribute above) and its depth-first search number to prioritize the expression.

Finally, the chosen expression is moved up to the boundaries of the current instruction group. The code motion process traverses the DAG from top to bottom starting from the group boundaries in search of the expression. To locate the original expressions, the intermediate sets saved on each basic blocks are used. When the expression is found, it is removed from the instruction stream, and the backward bottom-up traversal is used to updated invalidated availability sets. The bookkeeping copies are created at each join point on edges that do not belong to the current path traversed. The copies could be found and deleted when traversing along other paths, so e.g. when moving an instruction through a diamond to the dominating "if" block, first while traversing a path through "then" block a bookkeeping copy is created in the "else" block, and then the copy is found and deleted while traversing through "else" block.

Register renaming is supported by scheduling right-hand sides instead of whole instructions. An instruction is eligible for register renaming when it is a store to a register. For such an instruction, only its RHS participates in the scheduling process, i.e. when it is propagated up to the current fence, dependencies originated in its LHS are ignored. When choosing an RHS for scheduling, the suitable target register should be found, and the resulting expression will be scheduled as `best_reg = best_rhs`. When the new register is different than the old one, a register copy `old_reg = new_reg` will be left in place of original instruction.

To find the best register, it is needed to build the set of available registers which do not interfere with live ranges of other registers along the code motion paths. Therefore, the process of building such a set also requires traversing the code motion paths analogously to the actual moving stage.

Additional transformations such as substitution or speculation can be incorporated into the propagation stage. When an expression cannot be moved up through an instruction due to a dependence, one or more

transformations can be applied to eliminate the dependence. For example, `a = b + 5` can be moved through `b = c` as `a = c + 5`. If such transformation is made, the code motion process should undo it when searching for original instruction.

Selective scheduling supports software pipelining of innermost loops via manipulating fences. When a loop region is being scheduled, current set of fences represent "dynamic" backedges in the sense that code motion through those fences is not allowed. Therefore, at any time a scheduling region is actually acyclic. Moreover, each time a fence is advanced, a new edge in the region becomes the backedge, so that code motion through the edge that was previosly a backedge is now allowed (see Fig. 1, where current backedges are shown in red).

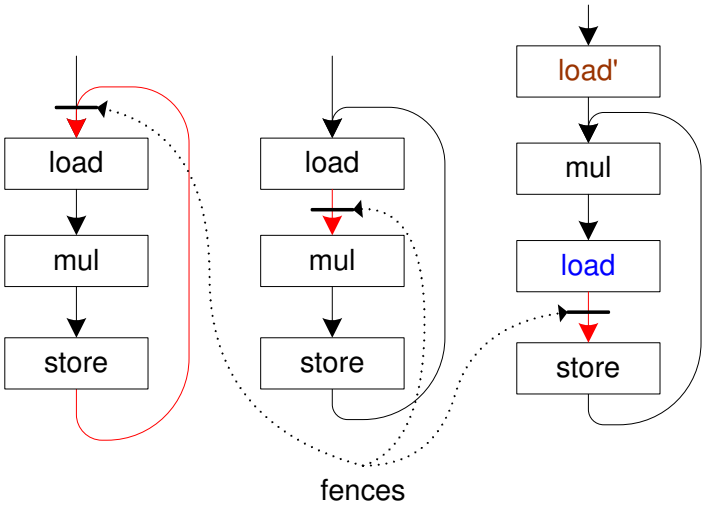

Figure 1: Software pipelining in the scheduler.

The actual pipelining happens when an instruction is being moved up along the real backedge of the loop. The bookkeeping copy of the moved instruction is created before the loop header, forming the prologue of the pipelined loop (shown in brown on the 'figure; the pipelined load is shown in blue). Analogously, the epilogue is created when a conditional jump to the beginning of the loop is being moved up, and bookkeeping copies are created on both targets of this jump.

This algorithm applies only to the innermost loops. Outer loops can be pipelined by treating the inner loop as a super-instruction, which is skipped when advancing fences. An instruction could be scheduled across an inner loop if the dataflow information of the loop is captured and saved, so that it can be used by the propagation routines.

3 GCC Implementation of the Selective Scheduling

In this section we will sketch the core of the implementation of the approach in GCC and the improvements we made to it, including handling of speculation / predication transformations and choosing heuristics.

We build the scheduling regions using the loop optimizer framework of GCC. The regions are formed

starting from innermost loops, so that when the inner loop is pipelined, its prologue can be scheduled together with yet unprocessed outer loop. The rest of acyclic regions are found using the same procedure that is implemented in GCC: the blocks that are not yet allocated to any regions are traversed in top-down order, and a block is added to a region to which all its predecessors belong; otherwise, the block starts its own region. The maximal number of instructions in a region is limited to 100; the limit for loops being pipelined is increased to 200.

There is a lot of controversy about the best placement of a scheduling pass in a compiler, and in GCC the situation is even more complicated. Currently, GCC has two scheduling passes, one before and one after register allocation, and the interblock scheduling is done only on the first pass. The second pass is a single block scheduling on all platforms except Itanium, where the second scheduler operates on extended basic blocks.

The problems with doing the scheduling before register allocation is that the intermediate representation at this time is so-called "non-strict" RTL, which means that there is no one-to-one correspondence between an IR instruction and an assembly instruction. On some targets such as Itanium, there is a lot of instruction patterns that get split on several instructions after register allocation. So, the first scheduling pass operates with inexact information about the target processor.

Due to this, we have decided to run the selective scheduler after register allocation, with the old scheduler still being run before register allocation. In this environment, register renaming is needed to help avoiding false dependencies. We have also tested the scheduler placing it before register allocation, but the initial attempts showed that the mechanism to control increasing register pressure is needed, and we didn't have time to implement such mechanism in the course of this project.

3.1 Implementing Instruction Transformations

While scheduling, instructions can undergo different transformations, which can be classified as *local*, i.e. ones that happen during propagation, and *global*, i.e. ones that require the knowledge of code motion paths. Substitution and speculation are local transformations, while register renaming and bookkeeping are global ones.

Instructions are divided into types depending on whether they allow a particular transformation to be performed or not. For local transformation this is checked by a target-dependent code (e.g. only loads are allowed for speculation). For register renaming, only *separable* instructions (which have LHS and RHS) are allowed. We also need to distinguish between the dependencies originated in LHS or RHS of an instruction to be able to discard dependencies from LHS. For this purpose, dependence analysis of GCC is modified so that when traversing an instruction, it marks the dependencies found appropriately. When a dependence is found, the analysis uses callbacks that may be defined by the scheduler. The callbacks are utilized both for propagating instructions up while determining dependencies and for finding out whether instruction's operands are ready.

For bookkeeping code support, we need to distinguish *unique* instructions, i.e. the ones that cannot be copied due to unpredictable side effects (inline assembly /calls). The unique instructions are propagated through join points anyway and filtered out on the choosing stage when one of the following happens:

- The destination (fence's) basic block does not dominate the source (instruction's) basic block;
- The instruction is not available on some path from the destination block to the source block.

We check the first condition by tracking the source basic block of expressions during propagation. When joining two expressions corresponding to unique instructions, we reset the source block value when they are

different. The second condition roughly corresponds to situations when `spec` attribute is nonzero. However, there could be situations when an instruction moved e.g. from 'then' block to 'if' block, in which case `spec` will be equal to one, but the code motion is still possible. We plan to handle this by additionally requiring that an expression should pass a joint point to be filtered out.

3.2 Choosing the Best Instruction

We are using a number of heuristics to prioritize the available expressions. The basic heuristics calculates the priorities by multiplying critical path length of expression and its *usefulness* (which shows the probability of executing the expression relatively to the current fence). Usefulness is calculated during propagation similarly to `spec` attribute. Additionally, the following heuristics are used to break ties:

- Jump instructions are preferred to non-jump ones on the grounds that scheduling jumps means fewer speculative instructions in the future (i.e., when a jump is ready for scheduling, other ready instructions are from other blocks and therefore speculative).

- Instructions that were scheduled fewer times are preferred. This is because scheduling an instruction more than one time means that it was moved from a previous loop iteration, and instructions moved from closer loop iterations are preferred, first, to let the algorithm terminate, and second, not to pipeline too much. This means we are only pipelining in the "holes" left from scheduling instructions from the current loop iteration.

  It should be noted than the original approach suggests not increasing `spec` attribute when moving an instruction along a back edge to encourage pipelining. We have tried this heuristics and got a performance drop. It should be possible that more work put in selecting instructions for pipelining will allow this heuristics to work though.

- For speculation, instructions with better probability of successful speculation are preferred. The code to implement this is shared with the old instruction scheduler of GCC, which has been supporting speculation since 2006 (see [Belevantsev06]). Also, speculative checks get lower priority than other instructions.

All these heuristics are computed on the fly during propagation. However, it is much harder to use heuristics based on "global" instruction features (similarly to global transformations we mentioned earlier), as they depend on the code motion paths of an instruction. Examples of such features include estimating profitability of bookkeeping code and of register renaming.

As we maintain the modeled processor state only in fences, we cannot estimate the influence of bookkeeping copies on the schedule of their blocks. Moreover, we know neither the places at which the copies will be created nor their number. While the former cannot be fixed easily in the selective scheduling, the latter can be precomputed when a set of available destination registers is computed for an expression. However, it is unclear whether prioritizing operations based on a number of bookkeeping copies they will create will give any benefit.

Register renaming and speculation are two examples of a transformation that leaves another instruction in the place of original one – either a speculative check or a register-register move, or both when they are combined. In such situation, the latency of this extra instruction should be accounted for when it is determined whether the transformation is desirable. For example, register renaming is used in the selective scheduling with assumption that register-register moves are cheaper than most other operations. However, this is not the case for Itanium, as a move between FP registers costs 4 cycles, which equals to the latency

of some other floating-point instructions such as addition. Similarly, most integer instructions take just one cycle to complete, which makes renaming them useless. [1]

We use a two-step approach for limiting register renaming. First, we do not allow renaming of instructions whose latency is less than two cycles. For the second part, a simple solution is do not allow renaming for instructions whose latencies are smaller or equal to the latency of a reg-reg move of appropriate register class. I.e, let $L(i)$ be the latency of instruction i, and let m be the corresponding `old_reg=new_reg` move, then renaming is allowed when $L(i) > L(m)$, as we definitely will save $L(i) - L(m) > 0$ cycles. However, this approach is very imprecise, as we assume that without renaming i will not be able to move higher from its current position.

Let T be the cycle on which i can be issued without renaming, and $T - d$ be the cycle on which i is issued with renaming, i.e. we have lifted i up by d cycles via renaming. Let E_r and E be the earliest cycle on which a use of i could be scheduled. Renaming is profitable when $E_r < E$, because then uses of i can be scheduled earlier. Given that $E = T + L(i)$ and $E_r = T - d + L(i) + L(m)$, profitability is achieved when $d > L(m)$, i.e. the instruction is moved up by some cycles more than the latency of the move instruction [2].

Using the above formula requires some effort. First, a latency of a move instruction depends on the register we choose for renaming, so we would like to choose a register with the cheapest move operation. In terms of GCC, this will usually be the register from the same register class as the original one. When a register from a different class is chosen, it could well be that the latency of renamed instruction will be different from the original one, i.e. $L(i_r) \neq L(i)$. In this case, the above formula will look like $d > L(m) + (L(i_r) - L(i))$. Second, it is not trivial to calculate d, because it boils down to estimating the number of cycles needed to execute instructions through which i was moved up, and there is a number of techniques to calculate upper and lower bounds of this number, e.g. [RimJain94]. Moreover, instructions that were originally below i can also move up between i and m, possibly increasing the interval between them. As a result, in the meantime we use the conservative approach described earlier. In the future, we can estimate d by looking at instructions gathered in the availability set together with i and assuming that all of these can be scheduled before m in addition to the ones i passed through, thus giving enough window for renaming to be profitable.

3.2.1 Estimating Profitability of Pipelining Code Motion

Everything we sketched above in this section applies both to the selective scheduler and pipeliner. The issue specific to the pipeliner is that it is hard to estimate the influence of the *pipelined* instructions (i.e. the ones that are moved from previous loop iterations) on the whole loop schedule. Consider the following typical loop.

```
load
some uses
jump back
```

The pipeliner starts from scheduling the code as usual, until it finds some holes that cannot be filled by the below instructions. At that point, it considers some instructions from above for pipelining, and if they fit, we see the following:

```
<...>                  <-- here was load
scheduled uses
```

[1] Except for the case when original instruction could not be scheduled in some instruction group due to an output dependence, but it can be scheduled with the new destination register.

[2] This calculation also assumes that resource constraints allow scheduling instructions on corresponding cycles.

```
scheduled load         <-- here was a free slot
yet unscheduled uses
jump back
```

We have used a free slot to pipeline a load there, but we have created a hole in the already scheduled code, which can potentially be filled in by one of the scheduled or unscheduled uses. We are not going to return back to the code to compact it further.

Another problem is that after pipelining instructions from the beginning of the loop get out of sync with their pipelined producers at the end of the loop. Consider the same example, in which the load has to become speculative in order to be pipelined, so we get the code like this:

```
speculative check     <-- here was load
scheduled uses
speculative load
jump back
```

The check here happens right after the jump on the beginning of a new iteration, so it is close enough to the load to stall. To fix this issue, more code should be scheduled between them, either at the beginning of the loop, or the load should be placed before some of the uses that are still working on the data from the previous iteration.

We try to improve the pipelined schedule by performing an extra scheduling pass over the pipelined loop, this time with pipelining, bookkeeping and renaming turned off. This technique aims at compacting the pipelined schedule by removing the holes shown in the first example above. We reschedule only affected basic blocks with very small scheduling window (9 instructions) to avoid too much increase of compile time. However, this does not address the second problem.

To avoid it, we use a simple heuristic approach to estimate the length of loop schedule and to reject obviously unprofitable renamings and speculations. Let c be the current scheduling cycle, l the length of the whole pipelined loop schedule, o – the cycle on which instruction i was scheduled first time, and $L(i)$ its latency. Then, the distance of code motion should be greater than instruction's latency to avoid stalls, i.e. $(l - c) + o > L(i)$. To estimate l, we use two heuristics. First, the largest instruction priority from the current availability set gives the rough estimation of the cycles needed to finish the loop, i.e. it is very probable that $l \geq c + \max_{i \in avset} Prio(i)$. Second, resource estimation is made by counting the number of instructions left to schedule. Empirically, we divide this number by $issue\_rate/2$ to obtain the number of cycles to the end of the loop. We are currently working on a more precise estimation that will count separately the instructions of a certain type (floating, integer, memory) left to schedule.

The last thing we plan to do is to take into account latencies of pipelined producers when scheduling consumers during rescheduling phase. For regular scheduling this is done by reflecting all scheduled instructions in a dependence context, so that later the dependence analysis can identify any dependencies with the currently scheduling instruction and check whether enough time had passed. In the rescheduling case, either the dependence context from the end of pipelining should be passed back to the beginning of the loop, or the new dependence context should be initialized with bookkeeping copies of pipelined instructions that are placed in the loop preheader:

```
copy of spec. load  // Init a dependence context
                    // with this copy

back:
speculative check
```

```
uses
speculative load
jump back           // Or take it from this jump
```

4 Improving Compile-Time Results

The selective scheduling algorithm is quite expensive in terms of compile time. In our experience, the main reason for this is the time needed for data dependence analysis and calculation of available registers. In the following subchapters we will describe our techniques for improving compile time numbers.

4.1 Caching Results of Dependence Analysis and Local Transformations

As noted above, the complexity of computing availability sets is concentrated in the `moveup_op` propagation routine, which decides whether an operation can be moved up through an instruction and in which form. Most of dependence analysis happens here. We have decided to perform the analysis on the fl for various reasons:

- Instruction dependencies may change after any transformation;
- An instruction may depend on different conditional jumps for different fences, as it will move along other code paths. This is complicated by the fact that GCC dependence analysis does not distinguish between control and data dependencies. As a result, it is hard to transfer control dependencies from a scheduled instruction to its bookkeeping copies;
- A dependence analysis should be used by both schedulers in GCC.

For the on-the-fly dependence analysis to be acceptable, we must avoid reanalyzing the same pairs of instructions many times during different computations of av sets. For this purpose, we maintain two bitmaps of `INSN_UIDs` for every instruction. The first bitmap indicates whether a particular instruction was ever moved up through this one, whereas the second bitmap caches the result of dependence analysis – whether a hard dependence was found or there is no dependence at all.

Local transformations itself are also costly. So in addition to the above bitmaps we use a hashtable that stores results of transformations for every instruction in the instruction stream. The standard hash of RTX expressions is used. The hashtable saves the information about the old instruction, the transformed one, and the speculative *status* of the transformed one (the status bits encode the type of speculation applied to the instruction and the probability of the applied speculation to be successful. The bits are used later when choosing between available speculative expressions.) The subsequent updates use this information without repeating transformations and generating excessive garbage.

We also record the history of all transformations applied to an expression in a vector saved in this expressions. This data is needed when undoing transformations in search for an original instruction. Ad-hoc undoing without this history is not just expensive, it does not allow e.g. moving `r33=sign_extend(r7)` through `r7=0`, as after substituting we get `r33=0`, and when undoing, there is no way to guess original instruction without saving it. This does not result in memory overhead, as an expression is usually transformed not more than 2-3 times.

4.2 Using Instruction Hashes

One of the operations that is heavily executed is comparison of two instructions. We use a notion of *virtual instruction*, or *vinsn*, to refer to either instruction or a whole or its RHS depending on instruction's type (separable, clonable, unique). Two vinsns are considered equal when they have the same type and pattern. To speed up the comparison, it is desirable to use instruction hashes. Fortunately, GCC already has a routine for hashing an expression. We hash the RHS for separable instructions and the whole pattern for others, recording the value as vinsn's hash. The hash is used for comparison as well as for storing history of transformations.

It should be noted that for speculative instructions we skip UNSPEC markers put on them to indicate speculativeness when hashing and comparing vinsns. This is done because we want to consider instructions with the same pattern but different types of speculation to be equal when mixing control and data speculation into one instruction. The similar thing happens in the original approach when for the sake of renaming we consider instructions with different patterns but same RHSes to be equal.

4.3 Optimizing the Memory Allocation Scheme

This problem is relevant to the GCC implementation of the selective scheduler and not to the approach itself, but we can see it arise in other implementations as well. There are a lot of short-lived objects, such as operations in av sets or regsets needed to store liveness information. These objects should be allocated using a pool scheme. In GCC, this approach is implemented in *allocpools*. The other problem is objects that by definition are allocated by a garbage collector, for example, instruction patterns. In this case, we are trying to reduce the amount of garbage as much as possible. For example, vinsns also act as a smart pointer, so they are shared between operations in different av sets. Caching of instruction transformations described earlier also falls in this category.

4.4 Speeding Up Work with Liveness Data

We have implemented two improvements in this area. First, in a scheduling region we save a valid liveness set on each instruction. This allows avoiding costly updates in large basic blocks when it is needed to traverse the block down and to propagate the temporary live set up to the needed instruction. It does not increase memory usage much as there is no more than 100-200 instructions in a region.

Second, we avoid register renaming when it is not needed. This is because the process of traversing code motion paths is very costly. We try to calculate whether a target register (the one in LHS) of an expression is not clobbered on the way up on the fly instead of traversing code motion paths afterwards. For a single register, it is quite simple to check the liveness restrictions when merging availability sets of a block's successors. We don't get right some of the corner cases, e.g. when expressions with different LHSes and same RHS are merged, or when after substitution the transformed expression is merged with another one in an availability set. But for majority of expressions we know that the original register can be used and avoid excessive traversions. Also, we limit renaming only to two high-priority expressions, as benchmarks show it does not make sense renaming less important expressions.

5 Experimental Results

When developing the scheduler, we have implemented a number of small improvements to the IA-64 backend. Below is the list of most important ones.

- Default alignment for loops and functions is set to 32 and 64 bytes, respectively, to follow the parameters used in the Intel compiler.
- Stop bits are now placed after each cycle to avoid unneeded placement of long-latency instructions in the same instruction group, which leads in stalling the whole group in case some of the instructions are not ready. This was originally advised by Mark Davis from Intel.
- The cost of true memory dependence between floating-point loads and stores which are not likely to alias is ignored (but not for speculative operations).
- The number of bundles that contain stop bits in the middle are minimized when the best bundle configuration is determined. This is because GCC performs bundling on each extended basic block separately, and without this heuristics suboptimal bundling placement can happen around labels.
- A heuristic to reduce cache bank conflicts is implemented. It decreases priority of memory operations in case one has already been scheduled in the current instruction group.

Table 1: Results on the SPEC FP 2000 benchmarks.

Benchmark	Trunk	Branch	% to trunk	New scheduler	% to branch	% to trunk
168.wupwise	517	517	0.00	562	8.70	8.70
171.swim	728	705	–3.16	761	7.94	4.53
172.mgrid	568	582	2.46	613	5.33	7.92
173.applu	500	532	6.40	526	–1.13	5.20
177.mesa	758	746	–1.58	763	2.28	0.66
178.galgel	794	790	–0.50	791	0.13	–0.38
179.art	1994	2018	1.20	2047	1.44	2.66
183.equake	513	517	0.78	569	10.06	10.92
187.facerec	974	987	1.33	985	–0.20	1.13
188.ammp	766	766	0.00	768	0.26	0.26
189.lucas	843	859	1.90	851	–0.93	0.95
191.fma3d	535	531	–0.75	516	-2.82	–3.55
200.sixtrack	296	315	6.42	323	2.54	9.12
301.apsi	599	602	0.50	612	1.66	2.17
Geomean	673.19	680.19	1.04	696.83	2.45	3.51

The experimental results shown in Table 1 are obtained with these heuristics turned on, `-O3-ffast-math` flags turned on, and disabled generation of auto-increment instructions (`-fno-auto-inc-dec`). We compare three runs: the old scheduler in GCC trunk, the old scheduler in our branch (with the backend improvements turned on), and the new scheduler in our branch. The scheduling window is set to 50 instructions. The results show that for floating-point benchmarks our backend improvements give 1% improvement, and the scheduler itself gives 2.5% improvement. The SPEC INT results are neutral, as we didn't do tuning on the integer benchmarks.

6 Modulo Scheduling Improvements

Swing Modulo Scheduling [Hagog2004, Llosa2001] is an algorithm for software pipelining. It exploits instruction level parallelism by overlapping operations from different iterations of the loop so they can be executed in parallel. The SMS algorithm has the following steps. First, the Data Dependence Graph

(DDG) is built for the loop. It's nodes represent loop instructions, and edges represent their intra- and inter-loop dependencies. Then, the *Minimum Initiation Interval* (*MII*) is calculated. The *MII* is the minimal number of cycles that a single iteration of the loop takes to execute. Then, SMS tries to schedule instructions of the loop in *MII* cycles, satisfying all dependence and resource constraints. If it is not possible with the *II* chosen, the algorithm starts over with the increased *II*, iterating until it succeeds or *II* exceeds the number of cycles in which the original loop could be scheduled.

In GCC modulo scheduling is implemented as a backend pass, which executes just before the first instruction scheduler. It generates the schedule for loops that it can handle (it requires the target loop to be a single-block counted loop) and marks them so the instruction scheduling pass won't reorder instructions inside these loops. To make SMS implementation in GCC work on IA-64, it required a minor fixes in code generation, namely, the fix for profitability check and correct generation of prologue instructions.

To perform effectively, SMS needs precise data dependency information from the compiler, including memory reference dependencies and distances between them in the iteration space. In GCC, there are two dependence analyzers, on the Tree SSA (high level) and the RTL (low level), respectively. For a given loop, the analyzer builds a vector of data references and a vector of dependence relations. A data reference contains links to a memory reference, a container statement and other memory attributes. A dependence relation contains the data references it links, its type, distance vector, direction vector, and subscripts information. The Tree SSA dependence analysis is used by the vectorizer and by the loop linear transformations.

The RTL array data dependence analyzer was written specifically for the swing modulo scheduling implementation in GCC. The analyzer builds a data dependence graph (DDG) for a given basic block. The DDG is represented as a vector of nodes. Each DDG node contains vectors of incoming and outgoing dependence edges, sets of successors and predecessors of the node in the DDG, and the containing instruction. Each DDG edge, analogously to the Tree SSA analysis, contains source and destination nodes of the edge, a dependence type, an edge latency, and a distance. Additionally, the edges that are going to/from the same node form a linked list analogously to control flow edges. The analyzer uses scheduler dependence analysis to build intra-loop dependencies and the data flow engine to build inter-loop dependencies.

Since SMS operates as an RTL pass, it can only use directly the results of the RTL dependence analysis, which has the following deficiencies:

- A distance of inter-loop dependencies is not calculated and is set to one conservatively. This limits the SMS implementation to interleaving instructions from neighboring iterations.
- Intra-loop dependencies are calculated using RTL alias analysis, which is not always able to disambiguate array references (especially on architectures that lack address displacement, like IA-64).

These problems may be avoided if the data dependency information is propagated from the Tree SSA to the RTL level[3].

To bring precise data dependencies information to modulo scheduling pass, we save dependence data for each memory reference as it is computed on Tree SSA level. This is done in the end of loop optimization passes on Tree SSA. For each loop nest in a function, we start from the innermost loop and calculate data dependencies for this loop. Then data references and data relations are saved in corresponding hashtables. When the data reference is already known for a given tree, i.e. it was calculated in the inner loop, we do not recalculate the reference to retain the information from the innermost loop. Then, upon generation of RTL, a link to this information is attached to the RTL memory references' attributes. The modulo

[3]The technical details of the process of data dependence propagation and its mapping from Tree SSA to RTL are beyond the scope of this paper and is described in details in [Melnik2007].

scheduling pass also has been modified accordingly so it can make use of better dependence data (including interloop dependencies) available through propagation from the higher level.

Since the rest of optimization passes are not aware about the propagated dependence information and may arbitrarily change RTL and Tree-SSA code without changing the attached dependence information, it required us to write a data verifier to ensure its correct propagation till the modulo scheduling pass. Using the data verifier we were able to identify those optimization passes that broke consistency of the propagated information and fix them, where applicable.

6.1 Testing SMS on SPEC CPU2000

We have tested our modifications to the modulo scheduling implementation on GCC mainline with SPEC CPU 2000 testsuite. The baseline configuration has `-O2 -ffast-math` flags enabled. We have also tested combinations of the following flags: `-fno-auto-inc-dec`, which works around the inability of SMS to work on loops with post-increment instructions; `-fmodulo-sched`, which enables SMS itself; and `-fexport-ddg`, which enables the exporting information and its uses. The results are summarized in the tables, which all have the same format: the first column shows the baseline results, and the subsequent columns show results with some flags added relative to this baseline.

Benchmark	-O2 -ffast-math -fno-auto-inc-dec	-fexport-ddg	-fmodulo-sched	-fmodulo-sched -fexport-ddg
168.wupwise	511	0.00%	1.57%	1.96%
171.swim	709	–0.14%	6.91%	6.77%
172.mgrid	264	0.00%	0.00%	0.38%
173.applu	431	0.23%	–0.23%	0.00%
177.mesa	729	–1.78%	–2.88%	–3.84%
178.galgel	815	–0.12%	–0.37%	11.04%
179.art	1927	–0.10%	–0.16%	0.00%
183.equake	459	0.00%	0.00%	0.00%
187.facerec	982	–0.20%	–0.51%	–0.51%
188.ammp	771	0.13%	–0.26%	0.00%
189.lucas	830	0.36%	0.00%	0.36%
191.fma3d	493	1.22%	0.00%	2.23%
200.sixtrack	182	–1.10%	–0.55%	–1.10%
301.apsi	554	0.00%	0.36%	1.81%
Geomean	594	–0.17%	0.17%	1.18%

Table 2: Results of the SMS improvements patch on SPECfp2000 benchmarks: DDG export, modulo scheduling and modulo scheduling with exported dependency information compared with gcc baseline.

For the SPEC FP benchmarks (see Table 2) the average speedup of modulo scheduling with DDG export compared to gcc baseline was 1.18%. We have analyzed some of the SMS successes, namely galgel with use of exported information and wupwise with -fmodulo-sched-allow-regmoves option which allows SMS use register moves to solve the overlapping live ranges issue. In both cases, SMS behaves as expected, i.e. the new information allows SMS to pipeline a loop.

We have also looked at some of the loops which SMS is supposed to pipeline but it failed. To aid this, we have used the Intel compiler version 10.0 for the reference. We have compared the dump reports created by SMS and the reports created by SWP of the Intel compiler. This analysis confirmed that in many cases an inaccurate dependence information is the cause of failure for SMS to pipeline loops. Another problem that was found for the art benchmark is that SMS fails to pipeline the loop because its minimal *II* is too high, whereas Intel's compiler doesn't miss the correct *II* by estimating minimal *II* better. The preliminary

result of the analysis of this problem is that Intel compiler performs better alias analysis.

SPEC INT	base	no auto-inc	ddg export	sms, no auto-inc	sms, no auto-inc, ddg export	sms, ddg export/ base	ddg export/ base	ddg export, sms/ export
164.gzip	680	679	679	681	681	0.15%	-0.15%	0.29%
175.vpr	801	800	801	800	799	–0.25%	0.00%	–0.25%
181.mcf	699	696	701	702	699	0.00%	0.29%	–0.29%
186.crafty	880	861	880	870	871	–1.02%	0.00%	–1.02%
197.parser	676	677	676	679	678	0.30%	0.00%	0.30%
253.perlbmk	803	801	803	809	813	1.25%	0.00%	1.25%
254.gap	588	592	586	597	598	1.70%	–0.34%	2.05%
256.bzip2	715	715	790	718	794	11.05%	10.49%	0.51%
300.twolf	1012	1007	1009	1002	1008	–0.40%	–0.30%	–0.10%
Geomean	752.51	750.17	760.47	753.71	762.76	1.36%	1.06%	0.30%

Table 3: Results of the SMS improvements patch on SPECint2000 benchmarks: DDG export (-fddg-export), SMS (-fmodulo-sched) and no autoincrement (-fno-auto-inc-dec) options in various combinations compared against the gcc baseline (-O2 -ffast-math).

As can be seen from Table 3, the best results for SPEC INT are achieved by the export patch alone thanks to the bzip speedup[4]. Additionally enabling modulo scheduling only slightly improves perlbmk, gap, and bzip. Results with and without auto-increment instructions vary, but the overall picture stays the same.

In average, testing on a SPEC CPU 2000 testsuite has shown 1.2% improvement for both SPEC FP and SPEC INT.

In the future, the following tasks could be completed to improve SMS implementation in GCC: add support for auto-inc-dec instructions, improve do-loop patterns recognition when finding loops to pipeline, and perform more comparisons with Intel compiler to identify loops that could be handled by SMS in GCC and determine what should be fixed to achieve that.

7 Conclusions

In this paper, we presented recent progress on the scheduling work of ISP RAS for the Itanium GCC. The selective scheduler is implemented on the sel-sched branch in GCC repository and is being prepared for inclusion into mainline GCC. It produces 3% mean speedup (up to 10% on certain tests) on SPEC FP 2000. Also, our work on the modulo scheduling implementation fixes gives 1% speedup (up to 5% on certain tests) for the same SPEC FP 2000 benchmark. We plan to continue working on improving the scheduler and software pipeliner in GCC for the Itanium architecture.

8 Acknowledgments

We'd like to thank Vladimir Makarov from Red Hat for extensive consulting of this project. The insights to instruction scheduling and GCC world provided by Vlad are invaluable. We also thank Daniel Berlin,

[4]There were three tests (176.gcc, 252.eon, 255.vortex) that were not tested due to the deficiencies in that SPEC setup and configuration, and also due to unrelated bug in gcc version we used.

Zdeněk Dvořák, Jan Hubička, Diego Novillo, Sebastian Pop, and James Wilson for giving helpful comments to this and other GCC work that we're doing. And we'd like to thank HP, Intel, and Gelato Federation for their attention to Itanium GCC.

References

[AburtoBenchmarks] Alfred Aburto's system benchmarks. Could be found at `ftp://gd.tuwien.ac.at/perf/benchmark/aburto`

[Belevantsev06] Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, and Dmitry Zhurikhin. An interblock VLIW-targeted instruction scheduler for GCC. In *Proceedings of GCC Developers' Summit 2006*, Ottawa, Canada, June 2006.

[GCCInternals] `http://gcc.gnu.org/onlinedocs/gccint`

[Ebcioğlu88] Kemal Ebcioğlu. Some design ideas for a VLIW architecture for sequential natured software. In *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing)*, North Holland, Amsterdam, 3–21, 1988.

[EPIC] Michael S. Schlansker, B. Ramakrishna Rau. EPIC: An Architecture for Instruction-Level Parallel Processors. HP Laboratories Palo Alto Technical Report HPL-1999-111, February 2000. `http://www.hpl.hp.com/techreports/1999/HPL-1999-111.pdf`

[Makarov03] Vladimir Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC. In *Proceedings of GCC Developers' Summit*, Ottawa, Canada, June 2003.

[Melnik2007] Dmitry Melnik, Sergey Gaissaryan, Alexander Monakov, Dmitry Zhurikhin. An Approach for Data Propagation from Tree SSA to RTL. GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, Brasov, Romania, September 2007.

[Moon97] Soo-Mook Moon and Kemal Ebcioğlu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. ACM TOPLAS, Vol 19, No. 6, pages 853–898, November 1997.

[RimJain94] M. Rim and R. Jain. Lower-bound performance estimation for high-level synthesis scheduling problem. IEEE Trans. CAD 13, 451-458. 1994.